The Serverkernel Operating System

The Serverkernel Operating System

Jon Larrea
University of Edinburgh
s2004865@ed.ac.uk

Antonio Barbalace
University of Edinburgh
abarbala@ed.ac.uk

ABSTRACT

With the idea of exploiting all the computational resources that an IoT environment with multiple mostly-idle interconnected devices offers, the *serverkernel* is presented as a new operating system (OS) architecture that blends ideas from Unikernels and RTOSes. These ideas are further mixed with a FaaS-like programming model to provide a server in which a user can remotely offload computations and get the result. Such OS architecture is minimalistic – a bare-metal OS in which only drivers for CPU, network, and accelerators are required in order to provide service.

To demonstrate the advantages of the *serverkernel*, jonOS, an open-source C implementation of this architecture for Raspberry Pi, is provided. Compared with traditional OSes used in IoT devices, the *serverkernel* achieves an improvement ratio of 1.5 in CPU time, 2.5 in execution time, and around 9 times better in network processing.

KEYWORDS

Operating Systems, Serverkernel, IoT, Unikernel, FaaS

ACM Reference Format:
Jon Larrea and Antonio Barbalace. 2020. The Serverkernel Operating System. In *3rd International Workshop on Edge Systems, Analytics and Networking (EdgeSys '20), April 27, 2020, Heraklion, Greece.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3378679.3394537

1 INTRODUCTION

In the last years, the demand for IoT devices has grown exponentially, stimulating vendors to broaden the offer of available devices on the market. With more IoT devices on the market, vendors compete for higher performance, smaller physical sizes (miniaturization), lower power consumption, etc. while the per-device price dropped down. Thus, IoT devices are today part of our life; they can be found in our city (e.g., intelligent hospital), workplace, home, and on our body (e.g., smartwatch). As a consequence, there are research and development to further make IoT devices part of our lives, including their integration in wearable and textiles [9, 16, 30], as well as into medical devices and prothesis [17, 28], which are supported by innovative production mechanisms, such as replacing the traditional silicon wafer with flexible materials [7]. Moreover, as many of these devices are battery-powered, but batteries cannot easily be charged, energy-harvesting technologies, which aims to implement power-independent platforms that can reap energy from movements or the environment, are at raise [20, 22, 27, 29].

Now we have an uncommon scenario with many devices attached to everyday objects. No one gets surprised when, at the shop, familiar items like speakers or lamps come with integrated voice assistant capabilities (e.g., Amazon Echo with Alexa support)! From a compute capacity perspective, this scenario opens the door to a world that provides a broad set of network interconnected processing units that remains the majority of the time in idle status. IoT lamps and lighting devices prove the truthfulness of this statement: their integrated processing units, commonly part of a System on Chip (SoC), leaves the idle status only when a new turn-on/turn-off command arrives from the WiFi chip. Ergo, creating a substrate of compute devices that can be opportunistically exploited to offload computations – especially when they are not energy-constrained, or have enough power to still correctly functioning for what they have been built.

At this point is where this research work comes into the scene, providing a high-performance solution to delegate load on mostly-idle IoT and generic embedded devices. The *serverkernel* is presented as a new OS architecture that takes advantage of this multi-device scenario allowing remote computation offloads, efficiently and securely. This new OS architecture mixes ideas from Unikernels, Function as a Service, and Cyclic Executive.

`jonOS` is the result of implementing the *serverkernel* OS on real hardware targeting high performance. After exhaustive benchmarks, `jonOS` demonstrated better performance of using traditional operating systems, namely Linux, in the described targe scenario.

2 BACKGROUND

Minimal OSes. Recently, there has been a returned interest in minimal operating systems with Unikernels [18] in the domain of cloud computing, and little real-time OSes in the domain of IoT [5, 6]. These fundamentally differ from the well documented traditional multi-user/task OS architectures described in [25] – which Linux [13] is a reference implementation. Linux has many variants that provide a complete solution for all kinds of scenarios, including embedded systems [31]. Thus, it is preferred in several deployments because it may reduce the time to market, and also because it is backed by a large development community that makes it easy to find people trained on Linux.

Unikernels are somewhat derived by the Exokernel OS design [14]. Such design strives to provide the closest access to hardware resources to the application itself, enabling the application developer to make low-level decisions. In an Exokernel OS most of the operating system services are deployed into a user-space library, called *libOS*, which is linked with the application itself. A Unikernel is quite similar; in fact, it also provides a *libOS* that is linked with the application itself. Of the *libOS*, only the functionalities that are strictly required by the application to run are used, the others are discarded during the compilation process – reducing the trusted computing base (TCB), and improving security [19]. Thus, Unikernels are tailored to fit the specific needs of an application. Additionally, the application and the *libOS* run in the same address

EdgeSys '20, April 27, 2020, Heraklion, Greece

space, which is kernel-space for a Unikernel. Due to the single address space, a unikernelized application does not cross address-space boundaries and has direct access to (potentially) all devices on a platform – thus, it can access them with the maximum performance.

Besides the security and performance advantages brought by Unikernels, Unikernels were not designed to run on bare-metal, but atop a hypervisor that exposes standard virtualized devices. Running a hypervisor is not always an option on embedded devices [23], and when it is, it may introduce non-trivial overheads, not just because of the hardware but also due to the software [21]. Unikernels usually support a very narrow set of devices, hence are not a good fit for embedded devices, including IoT. Therefore, most IoT devices today adopt little real-time operating systems [26] (RTOS). Such OSes offer time guarantees for applications that need to process data as it comes to the device – i.e., producing an answer in a specific amount of time from when the data is received. However, most RTOS today are bloated with additional code to support multimedia, multi-tasking, etc., incorporating design traits from traditional multi-user/task OS architectures. Hence, less suitable for maximum performance.

Function as a Service. Function as a service (FaaS) is probably the most successful Cloud innovation so far. FaaS is a category of cloud computing services that provides a platform allowing customers to develop, run, and manage applications, but without the burden of building and maintaining the infrastructure [10, 24].

With FaaS, a developer identifies language code-level functions (or hierarchy of them) to be executed on event triggers, on a machine in the cloud data-centre. Thus, FaaS enables the execution of modular pieces of application code on a remote machine when an event, e.g., a service request, or data ingestion, happens. FaaS application code can access cloud storage and maybe, provided with input data. Where and when the application code is executed is completely under the control of the service provider. Major examples are AWS Lambda [2] and Azure Functions [3].

3 DESIGN AND ARCHITECTURE

The proposed OS is based on the design principle of extreme minimality, for high performance and energy conservation, to the detriment of being general-purpose, but without sacrificing security. It targets IoT and generic embedded devices with a network connection of any kind, including Bluetooth, WiFi, and Ethernet, which are mostly idle or have computing and network resources that are mostly idle. The proposed OS controls such idle hardware resources whenever they become available, and make them exploitable by other computing devices on the network, which can use them to offload computing tasks. Once the OS gets in control of the hardware resources, it acts as a server waiting for a request on a network interface, but in this case, a request consists of a compute task to execute and the associate data. When the task is received by the OS, it is executed, and the eventual result of the computation sent back. It is essential to highlight that the proposed OS does not define any computing task by its own; the task is defined remotely by the client. Even so, we don't impose the data to be shipped by the client (see below).

The *serverkernel* is a single-address space mono-task OS. The OS kernel and the application code – which has been received via the network, reside both in kernel-space to avoid context switching overhead. At the same time, to do not expose the hardware to security risks, application code is run in a shielded environment, and a watchdog periodically checks for OS liveliness. For the sake of minimality, the *serverkernel* offers a limited set of functionalities. Only device drivers for network cards, CPU, and accelerators (included security engines) are necessary to achieve high-performance executions.

This could be thought of as a Unikernel, which is also single-address space, but its main functionality is a server. However, a *serverkernel* cannot be optimized according to just one application's requirements because it has been designed to run generic application code. From the functionality perspective, the proposed OS sounds similar to a FaaS runtime, but the latter runs in user-space, not integrated within the OS kernel. Moreover, the dispatching of application code to the *serverkernel* is directed by the client itself, while in FaaS it is the responsibility of the data-center provider. Hence, the serverkenel borrows concepts from Unikernels and FaaS, and further combines them within a cyclic executive [4] state machine which, after receiving a new execution request in the idle state, goes to the running state and, after the execution, goes again to the idle state.

The proposed mono-task execution model allows for one application running at the time. Although this is acceptable for an embedded devices dedicated to run a *serverkernel*, most IoT and generic embedded devices are shipped with their own proprietary software. Therefore, we envision a *serverkernel* to either run atop an hypervisor [23] together with the proprietary software, or cooperative coexist with the proprietary software [11, 12, 32]. In the first case, the hypervisor sandboxes any eventually malicious running on the *serverkernel*. In the second case, additional measures should be taken; including auto-resetting if the system detects that it has been attacked.

From the client perspective, the *serverkernel* requires the application code to be offloaded to be compiled in a specific format. Thus, a developer, or the client software, should be able to finalize the application code in that format.

3.1 Operating Principles

In order for remote devices to run application code on a *serverkernel* they have to (1) identify a *serverkernel* that can be accessed from a network connection; (2) attest the *serverkernel* is not malicious; (3) compile the code in the format advertised by the *serverkernel*; (4) establish a secure connection; (5) send the code and eventual data to the *serverkernel*; (6) wait for the result, which will also drop the connection.

In order to be discovered by remote devices, a *serverkernel* sends broadcast messages on any of its network connections periodically when ready to accept incoming tasks for processing – different network technologies implement broadcasting differently. Remote devices would then attest the *serverkernel* is a genuine one, running on a genuine board. Thus, the client attempts remote attestation of the *serverkernel* [15]. During this phase, the *serverkernel* advertises its available hardware and software resources – this includes

the specification of all available processing units, and the offered processing time, as well as the accepted binary format(s). At this point, the remote device can compile, or finalize an intermediate representation, to one of the advertised binary formats. Then, it will establish a secure connection with the *serverkernel* to download the binary and associated data. When received, the *serverkernel* execute it, and send back the results of the computations. Before executing a binary, the *serverkernel* checks its integrity and loads it in memory with associated data – during this phase, it may perform some sort of runtime linking for symbols provided by the kernel.

4 IMPLEMENTATION

To prove the viability of the proposed OS architecture, we implemented an initial prototype capable of running bare-metal from scratch, and we called it `jonOS`. The prototype integrates most of the functionalities required by the *serverkernel* OS design, and it is a stepping stone to implement our vision. Although most functionalities are implemented, and application code and data offloading from remote devices works, the binary blob including code and data must be prepared manually – i.e. we did not add yet automatic code synthesis capability to any mobile OS framework, e.g., Android or iOS. Moreover, security-related features are under development.

`jonOS` is our open-source[1] implementation of the *serverkernel*. Currently, `jonOS` supports different low-cost ARM boards based on the BCM2835 SoC [1] – a mono-core ARM 6, including Raspberry Pi 1B and the Raspberry Pi Zero, which can be bought for as few as 5GBP [8] and it comes with WiFi connectivity! `jonOS` has been implemented in C (around 6100 LOC), and assembly language (around 850 LOC) to seek performance improvements. `jonOS` is modular, a module is an OS kernel device driver or library, and modules are grouped by their purpose. The most relevant *device driver* modules are the Serial (or UART), Network (or NIC), Screen, and CPU/GIC. Atop of these, there are the *manger* modules, such as network, debugging, and memory, which enable application code to run thanks to the modules in the *cyclic executive* and *standard library*. Figure 1 depicts this architecture. Additionally, `jonOS` comes with a compiler toolchain to simplify the creation of its binary blobs.

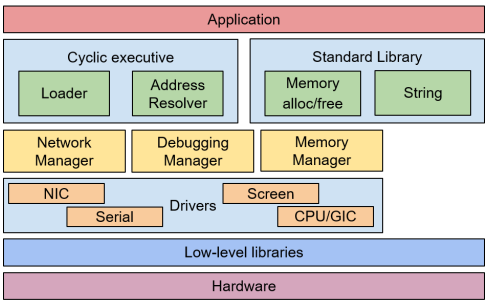

Figure 1: `jonOS` modular architecture.

Debugging Management. The Universal Asynchronous Receiver-Transmitter module (UART) is a serial communication device driver that uses the GPIO 14 port for the transmission and GPIO 15 port for the reception at 115200 bauds. This module has been implemented for debugging purposes in an early version of the operating system. However, the serial line for debugging purposes is a temporary solution, and the goal is to provide a remote alternative.

Network Manager. The network module has three main components: Ethernet driver, USB LAN9512 hub, and the network stack. To get access to most the devices in the Raspberry boards, it is necessary to do it through the LAN9512 USB hub that has a bandwidth of 100Mbps. This USB driver also includes a low latency diver to get access to the Ethernet driver that is connected to it. On top of this driver, a lightweight and high-performance UDP/IP stack has been implemented to reduce latency. This optimization has been possible thanks to a naive implementation of the IP header, which uses default values in fields such as type of service, time to live, or header length. This decision reduces the number of checks that the stack has to do to validate the header. Another unused field is the port number in the UDP header. Because there is only one running process at the time, the port number loses its utility (differentiate among processes). Despite, this implementation allows filter by port or allows traffic at any port. This stack uses a 16 entries ARP table to reduce times in the sending process to known hosts.

Screen. The screen module allows access to the HDMI port through the Videocore IV GPU using the mailbox protocol. This diver automatically detects the size of the connected screen and provides a POSIX "`printf`" function to display a text on it. The displayed text is configurable with different colours, backgrounds, and sizes. This driver also allows displaying BMP (Bitmap image) images.

Memory Management. The dynamic memory module provides a naive implementation of malloc and free functions. This module works using the same mechanism around "`sbrk`" function and "`brk`" pointer but with a single linked-list. The reason for that is that the complexity of a double linked-list is higher and eventually produces higher latency.

Standard library. The standard library provides a high-performance implementation of some of the most commonly used functions in the C library. Some of the functions are: memory management (e.g., "`memset`"), string management (e.g., "`strlen`") and types conversion (e.g., "`atoi`"). The improvements done in this module are related to memory management. As an example, "`memcpy`" function copies a chunk of memory of specific length from one position to another. This is done by copying every byte, pointed by a "`uint8_t`" pointer, from the origin to the destination address. This implementation calculates if the length parameter is module 2 and 4 and if so, then replaces the "`uint8_t`" pointer by "`uint16_t`" pointer or "`uint32_t`" pointer in order to reduce the number of memory access by 2 or 4 respectively.

Low-level libraries. In the low-level libraries set, there is a large variety of functions. The interrupt controller and cache utilities are two subsets of the functionality implemented in C. Other functions require to be implemented in ARM assembler because they are more frequently used. Even though the ARM1176JZFS (CPU integrated into the BCM2835 SoC) has floating-point support, there is no native

[1]The code is available at https://github.com/j0lama/jonOS .

division instruction. An efficient division routine is an example of functionality implemented in assembler.

4.1 jonOS Executable Binaries

As mentioned in the *serverkernel* requirements, every implementation has to provide a toolchain that allows a client to compile and deploy applications remotely on the *serverkernel* device. `jonOS` comes with a toolchain that allows remotely compile a payload (name received by the applications compiled with this tool-chain) and execute it sending it through the network. This toolchain requires the use of an ARM cross compiler to generate code runnable in the Raspberry. This is achieved by the GNU Toolchain `arm-none-eabi` that generates binaries (32 bits).

Executable Format. Because the serverkenel allocates the application in different addresses every execution, the code has to be generated as position-independent code (PIC). This guarantees that the code can be executed at different memory addresses. The second requirement pretends to provide access to the `.DATA` section variables regardless of the address in which the application resides. To do that, all the variables accessed in the `.TEXT` section has to be referenced through the IP/PC register, and all the variables are introduced in the `.TEXT` section. The last compiler requirement tries to provide more versatility to the programmers allowing them to use reserved function names such as "`main`" in is application/-function. Because the compiled application has to be a function to be loaded in the *serverkernel*, this option allows the programmer to call the function "`main`" without compiling it as an entry point of an ELF executable.

There is a set of Python tools included in `jonOS` that reduces the size of the payload. These tools extract from the generated ELF binary only the required sections, and then, a new binary structure is assembled with significant size reduction. An explanation of this process is shown in Figure 2, where only the `.TEXT` section and a particular `.RODATA` subsection, in which strings are allocated, are used to produce the `jonOS` executable.

For example, a function with a "`while(1)`" statement that locks the execution in an infinite loop is an example of this reduction. The generated ELF executable before the size reduction process is 612 bytes binary, whereas the generated Payload after this process is a 14 bytes binary. The reduction rate for this example is almost 44 times, but it is reduced in larger applications in which the bulk of the size resides in the `.TEXT` section.

Executable Loading. One of the most relevant features is how `jonOS` deals with the system call addresses in a simple but effective way. Because the Payloads are only compiled and not linked, the *serverkernel* OS has to provide a tool-chain to allow the client to use functions implemented in the system. As mention before, the main focus of this implementation is increased performance but keeping simplicity. The selected address resolver is a MAP structure in with the keys are integer variables, and the values are pointers to the associated function. To access this map structure, the kernel implements a function whose address has to be known by the client that, given a pre-defined number, return the address of the associated function. These numbers are all defined in the tool-chain library necessary to compile Payloads. The disadvantage of this

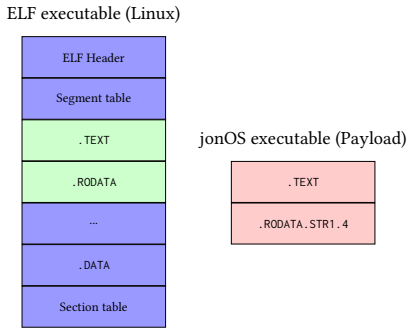

Figure 2: Binary structure comparison between ELF executable (Linux) and Payload (jonOS).

method is that the tool-chain has to have the address of this function hardcoded what implies that every kernel version requires a specific tool-chain or at least a specific address hardcoded on it. On the other hand, this method provides a *O(1)* performance because the MAP structure has been implemented as a Hash-MAP structure. Another advantage of this method is that the OS admin can define a set of functions that are only available for the Payload, increasing the security.

The client has to define the function as a pointer and call the address resolver function provided by the tool-chain with the number associated with that function, also defined in the tool-chain. In Figure 3 this process is illustrated.

```
1 #include "libkernel.h"
2
3 int function()
4 {
5   int*(*recv)(uint16_t, void *, size_t) =
        solveFunction(RECV);
6   ...
7 }
```

Figure 3: The procedure to get the effective address of "recv" function through the address solver library.

This method not only helps to reduce the time required to resolve the system call addresses but also helps to decrease the size of the payload because no extra code is added to the .TEXT section during compilation.

5 INITIAL RESULTS

The main objective behind this research is to defining a new OS architecture that takes full advantage of the idle resources available on IoT and embedded devices by delegating computational workloads over to them on demand. However, the same functionality can be implemented by traditional OSes. Therefore, in order to demonstrate the advantages of this new OS architecture, time measurements have been made on three different experiments. Trying to approximate this experiments to a real scenario, a Linux operating system has been chosen as a representation of the traditional OS deployed in commercial devices. The chosen Linux OS is Raspbian; a Debian fork, adapted to Raspberry Pi boards, that is extensively

used in this platform. Raspbian has been developed by Raspberry Pi Foundation, the same company that produces Raspberry Pi.

The experiments can be classified into two different groups: compute and network. The reason of that is because, in this scenario, where many devices are connected to the network in order to offload computational task on them, compute and network performance are the critical parameters in order to decide what OS architecture to use.

5.1 Compute Performance

To measure compute performance, CPU time and execution time have been used. CPU time is defined as the time that a specific task spends running on the CPU, without accounting for kernel or device access times. Execution time is defined as the time that a specific task takes until it ends. Thus, kernel and device access times are included. Note that the CPU time and the execution time in a mono-task operating system are the same because the operating system does not need to change context between processes.

As a benchmark for both measured times, a Message-Digest Algorithm 5 (MD5) hash calculation function has been used. The code, written in C, is identical for both operating systems. The benchmark consists of an increasing number of executions of the hash function in order to provide a significant load amount.

CPU-time. Only on Raspbian, two different approaches have been used to measure CPU time and real-time. For CPU time, the `clock()` function has been used to get the time before and after the task, and then calculate the elapsed time. For the real-time measurement, a PL-2303 converter has been used to notify through the UART when the task begins and ends. This latter method has been used to measure the CPU time/real time on `jonOS`.

Figure 4 shows the CPU time comparison for different values of N between 1 and 1000000, where N is the number of executions of the hash function.

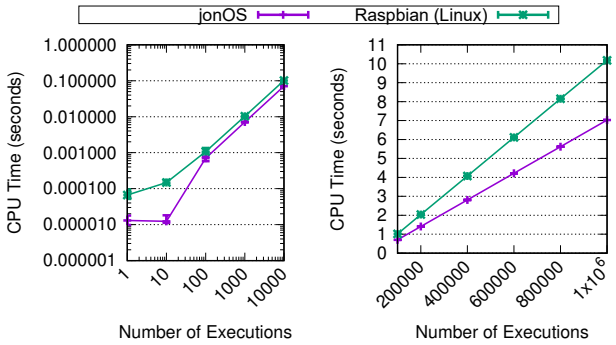

Figure 4: CPU time comparison using MD5 benchmark.

The graph reflects an improvement of 45.67%. This is because a multi-task OS has to change context between task; thus, it updates cache memories. In the Raspberry Pi context, the cache memories that need to be updated after a change of context are data cache L1, L2 and instruction cache. This delay is directly reflected in the CPU time.

Execution time. A TTL to USB converter connected to GPIO 14 and GPIO 15 has been used to measure the real-time. From an external computer, a start message is sent to both operating systems to start a timer. At the end of the execution, the operating systems send a message to the external computer that is waiting to stop the timer. Because the same Python script has been used for both operating systems, the additional time added by the script is not relevant.

As Figure 5 depicts, values of N between 1 and 1000000 has been used in order to provide different scenarios with a different workload.

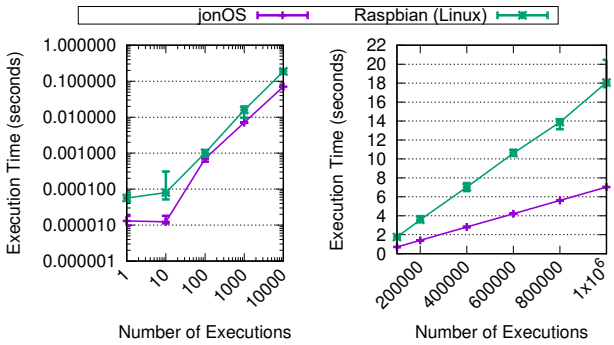

Figure 5: Execution time comparison using MD5 as benchmark.

For the real-time, `jonOS` reduces it in 62% compared to Raspbian (improvement ratio of 2.5x on average). This data reflects the performance difference between share resources among multiple processes (Raspbian) and dedicate all of them to only one. Furthermore, the algorithms implemented in the scheduler of the multi-task operating system may harm this task if a low priority is assigned to it. It is essential to highlight that, by default, in Linux, multiples processes are running after the boot process, and the task shall share resources with them.

5.2 Network Performance

One of the simplest ways to measure the network stack performance is by using an echo server. An echo server works by replying to the origin, with the message content received. For this experiment, a Python UDP client has been used. This client sends N number of messages to the server and waits until receiving all the echo packets. The measured time starts when the first packet is sent by the client and finishes when the last echo packet is received. A relevant factor that can affect the measured times is network architecture. Therefore, a point to point connection between the Raspberry and the computer that executes the client has been used. This network topology removes all the possible router bottlenecks in between of the hosts. Besides, the noise produced by other kinds of traffic is also removed.

The same echo server has been implemented in both systems but using each native network stacks.

Times for values of N between 1 and 100000 requests are shown in Figure 6.

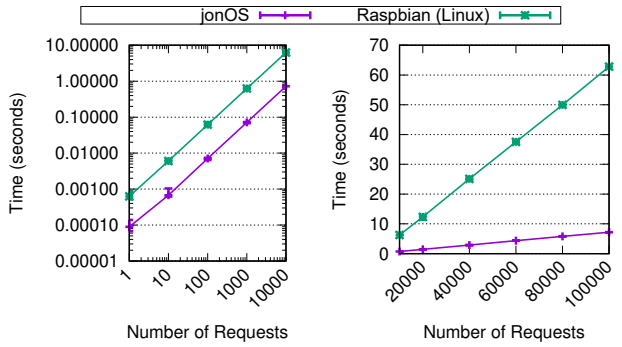

Figure 6: Network experiment time comparison using a echo server as benchmark.

As Figure 6 shows, performance up to 8.94x (8.53x on average) times better than Raspbian is given by jonOS. The reason for this time difference resides in multiple implementation/design aspects:

(1) Similarly to the hash calculation, the CPU and network stack are shared among multiple processes in Raspbian, and the echo server can be interrupted by a higher priority process.
(2) Because in the *serverkernel* there is only one running process, the operating system does not need to use the port to redirect the packet to the process bind to that port and reducing the time spent in the transport layer.
(3) When in a Linux system, a message is sent to the network using the socket library, the packet may wait until the output buffer gets enough size to be emptied by the operating system. In the current jonOS implementation, the packets are directly sent to the network.
(4) The stack used in jonOS is a high-performance stack whose objective is to reduce latency whereas Linux stack is a generic stack that focuses on reliability.

With the topology used in this experiment, it is not possible to determine if the times marked as jonOS times in the graph corresponds with jonOS network stack or this limit is reached by the computer that was running the client. This experiment guarantees that jonOS has a maximum latency ceiling in the values showed in the graph.

6 CONCLUSIONS

This work introduces the *serverkernel*, a new operating system architecture. The *serverkernel*, through the jonOS implementation, has proved to be a better alternative to traditional OSes for an environment in which a user wants to offload computations on remote IoT and generic embedded devices that are mostly idle. jonOS shows to be on average 1.5 times, 2.5 times, and 8.5 times faster than Linux in CPU time, execution time, and network processing.

REFERENCES

[1] 2012. BCM2835 ARM Peripherals. (2012). https://www.raspberrypi.org/app/uploads/2012/02/BCM2835-ARM-Peripherals.pdf
[2] 2020. AWS Lambda. (2020). https://aws.amazon.com/lambda/
[3] 2020. Azure Functions. (2020). https://azure.microsoft.com/en-gb/services/functions/
[4] 2020. Cyclic executive. (2020). https://en.wikipedia.org/wiki/Cyclic_executive
[5] 2020. FreeRTOS. (2020). https://www.freertos.org/
[6] 2020. Huawei LiteOS. (2020). https://www.huawei.com/minisite/liteos/en/about.html
[7] 2020. PragmatIC. (2020). https://www.pragmatic.tech/
[8] 2020. Rapsberry Pi Zero. (2020). https://thepihut.com/products/raspberry-pi-zero?src=raspberrypi
[9] ATHOS. 2020. Athos Training System. (2020). https://www.liveathos.com/
[10] Abel Avram. 2016. FaaS, PaaS, and the Benefits of the Serverless Architecture m. (2016). https://www.infoq.com/news/2016/06/faas-serverless-architecture/
[11] Antonio Barbalace, Binoy Ravindran, and David Katz. 2014. Popcorn: a Replicated-kernel OS Based on Linux. In *In Proceedings of Ottawa Linux Symposium (OLS '14)*.
[12] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. 2009. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP '09)*. 29–44.
[13] Daniel P. Bovet and Marco Cesati. 2002. *Understanding the Linux Kernel* (first ed.). O'Reilly.
[14] D. R. Engler and others. 1995. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles (SOSP '95)*. 251–266.
[15] Matthew Garrett. 2019. What does Remote Attestation buy you? (2019). https://www.linuxplumbersconf.org/event/4/contributions/295/attachments/374/608/What_does_Remote_Attestation_buy_you_.pdf
[16] Tuan [Nguyen Gia], Victor Kathan Sarker, Igor Tcarenko, Amir M. Rahmani, Tomi Westerlund, Pasi Liljeberg, and Hannu Tenhunen. 2018. Energy efficient wearable sensor node for IoT-based fall detection systems. *Microprocessors and Microsystems* 56 (2018), 34 – 46.
[17] C. Kulkarni, H. Karhade, S. Gupta, P. Bhende, and S. Bhandare. 2016. Health companion device using IoT and wearable computing. In *2016 International Conference on Internet of Things and Applications (IOTA)*. 152–156.
[18] Anil Madhavapeddy and others. 2013. Unikernels: Library Operating Systems for the Cloud. *SIGARCH Comput. Archit. News* 41, 1 (March 2013), 461–472.
[19] Anil Madhavapeddy and David J Scott. 2013. Unikernels: Rise of the virtual library operating system. *ACM Queue* 11, 11 (2013).
[20] Elizabeth Montalbano. 2016. Energy Harvesting, Low Power Consumption Are the Way Forward for IoT, Wearables. (2016). https://www.designnews.com/iot/energy-harvesting-low-power-consumption-are-way-forward-iot-wearables/212976763446132
[21] Vlad Nitu, Pierre Olivier, Alain Tchana, Daniel Chiba, Antonio Barbalace, Daniel Hagimont, and Binoy Ravindran. 2017. Swift Birth and Quick Death: Enabling Fast Parallel Guest Boot and Destruction in the Xen Hypervisor. In *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '17)*. 1–14.
[22] K. Z. Panatik, K. Kamardin, S. A. Shariff, S. S. Yuhaniz, N. A. Ahmad, O. M. Yusop, and S. Ismail. 2016. Energy harvesting in wireless sensor networks: A survey. In *2016 IEEE 3rd International Symposium on Telecommunication Technologies (ISTT)*. 53–58.
[23] S. Pinto, J. Pereira, T. Gomes, M. Ekpanyapong, and A. Tavares. 2017. Towards a TrustZone-Assisted Hypervisor for Real-Time Embedded Systems. *IEEE Computer Architecture Letters* 16, 2 (2017), 158–161.
[24] Mohammad Shahrad, Jonathan Balkind, and David Wentzlaff. 2019. Architectural Implications of Function-as-a-Service Computing. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '52)*. 1063–1075.
[25] William Stallings. 2001. *Operative Systems: Internals and Design Principles* (fourth ed.). Pearson.
[26] John A. Stankovic and others. 2004. Real-Time Operating Systems. *Real-Time Syst.* 28, 2–3 (Nov. 2004), 237–253.
[27] S. Sudevalayam and P. Kulkarni. 2011. Energy Harvesting Sensor Nodes: Survey and Implications. *IEEE Communications Surveys Tutorials* 13, 3 (2011), 443–461.
[28] Ashok Vaseashta. 2018. Roadmapping the Future in Defense and Security: Innovations in Technology Using Multidisciplinary Convergence. In *Advanced Nanotechnologies for Detection and Defence against CBRN Agents*. Springer Netherlands, 3–14.
[29] vesper. 2020. Voice interface devices are everywhere, including where there is no access to power outlets. (2020). https://vespermems.com/applications/smart-home-smart-office-iot/
[30] Gary M. Weiss and Md. Zakirul Alam Bhuiyan. 2019. *An Overview of Wearable Computing*. 313–349.
[31] Karim Yaghmour and others Jon Masters. 2008. *Building Embedded Linux Systems* (second ed.). O'Reilly.
[32] Gerd Zellweger, Simon Gerber, Kornilios Kourtis, and Timothy Roscoe. 2014. Decoupling Cores, Kernels, and Operating Systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*. 17–31.